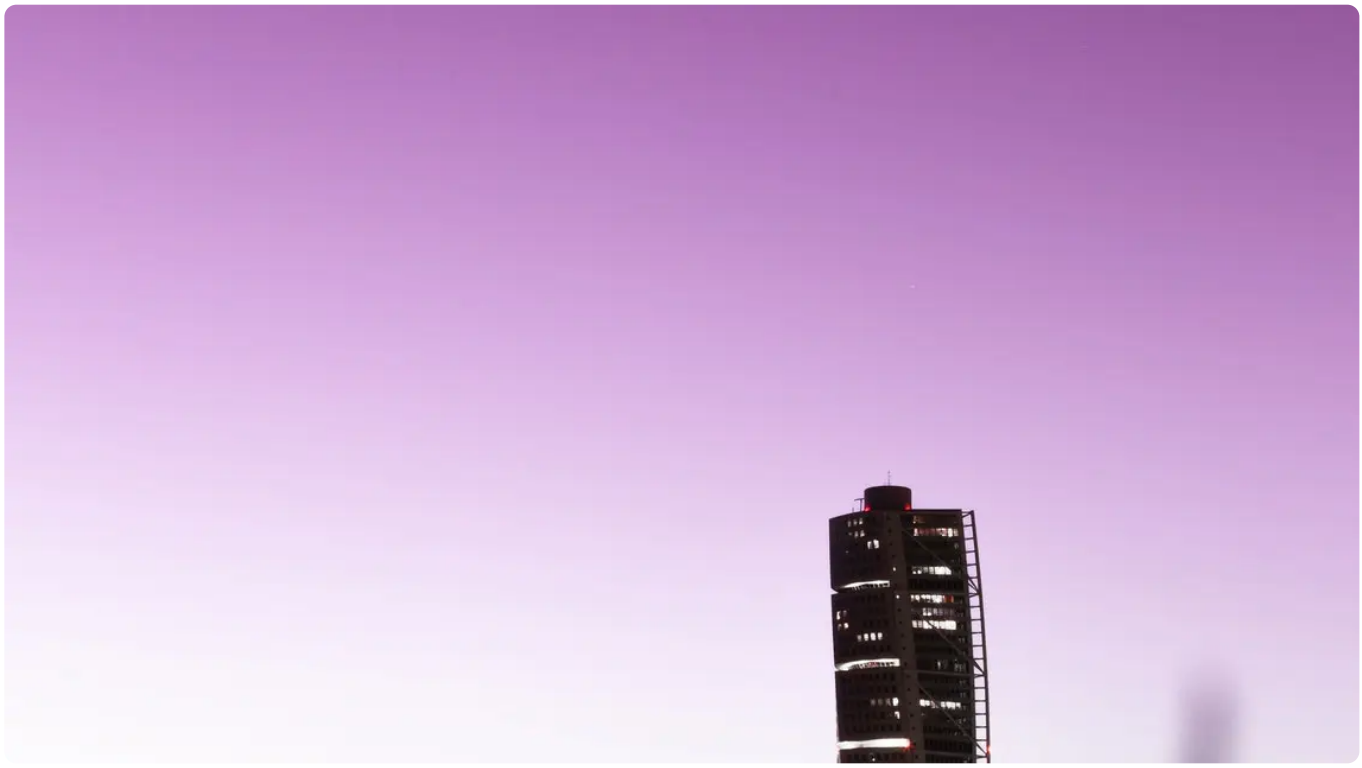
下载APP

08 | 基于TypeScript的虚拟机（一）：实现一个简单的栈机

2021-08-25 宫文学

《手把手带你写一门编程语言》 课程介绍 >



讲述：宫文学
时长 18:43 大小 17.16M

你好，我是宫文学。

上一节我们已经探讨了设计一个虚拟机所要考虑的那些因素，并做出了一些设计决策。那么今天这一节，我们就来实现一个初级的虚拟机。

要实现这个初级虚拟机，具体来说，我们要完成下面三方面的工作：

首先，我们要设计一些字节码，来支持第一批语言特性，包括支持函数、变量和整型数据的运算。也就是说，我们的虚拟机要能够支持下面程序的正确运行：

复制代码

```
1 //一个简单的函数，把输入的参数加10，然后返回
```

```
function foo(x:number):number{
    return x + 10;
}
//调用foo，并输出结果
println(foo(18));
```

第二，我们要做一个字节码生成程序，基于当前的 AST 生成正确的字节码。

第三，使用 TypeScript，实现一个虚拟机的原型系统，验证相关设计概念。

话不多说，开搞，让我们先设计一下字节码吧！

“设计”字节码

说是设计，其实我比较懒，更愿意抄袭现成的设计，比如 Java 的字节码设计。因为 Java 字节码的资料最充分，比较容易研究，不像 V8 等的字节码，只有很少的文档资料，探讨的人也很少。另外，学会手工生成 Java 字节码还有潜在的实用价值，比如你可以把自己的语言编译后直接在 JVM 上运行。那么我们就先来研究一下 Java 字节码的特点。

上面的用 TypeScript 编写的示例代码，如果用 Java 改写，会变成下面的程序：

复制代码

```
//实现同样功能的Java程序。
public class A{
    public static int foo(int a){
        return a + 10;
    }
    public static void main(String args[]){
        System.out.println(foo(8));
    }
}
```

我们首先把这个 Java 程序编译成字节码。

复制代码

```
javac A.java
```

这个文件是一个二进制文件。我们可以用 hexdump 命令查看它的内容。

复制代码

```
hexdump -C A.class
```

```
➜  06 hexdump -C A.class
00000000  ca fe ba be 00 00 00 3b  00 1f 0a 00 02 00 03 07  |.......;........|
00000010  00 04 0c 00 05 00 06 01  00 10 6a 61 76 61 2f 6c  |..........java/l|
00000020  61 6e 67 2f 4f 62 6a 65  63 74 01 00 06 3c 69 6e  |ang/Object...<in|
00000030  69 74 3e 01 00 03 28 29  56 09 00 08 00 09 07 00  |it>...()V.......|
00000040  0a 0c 00 0b 00 0c 01 00  10 6a 61 76 61 2f 6c 61  |.........java/la|
00000050  6e 67 2f 53 79 73 74 65  6d 01 00 03 6f 75 74 01  |ng/System...out.|
00000060  00 15 4c 6a 61 76 61 2f  69 6f 2f 50 72 69 6e 74  |..Ljava/io/Print|
00000070  53 74 72 65 61 6d 3b 0a  00 0e 00 0f 07 00 10 0c  |Stream;.........|
00000080  00 11 00 12 01 00 01 41  01 00 03 66 6f 6f 01 00  |.......A...foo..|
00000090  04 28 49 29 49 0a 00 14  00 15 07 00 16 0c 00 17  |.(I)I...........|
000000a0  00 18 01 00 13 6a 61 76  61 2f 69 6f 2f 50 72 69  |.....java/io/Pri|
000000b0  6e 74 53 74 72 65 61 6d  01 00 07 70 72 69 6e 74  |ntStream...print|
000000c0  6c 6e 01 00 04 28 49 29  56 01 00 04 43 6f 64 65  |ln...(I)V...Code|
000000d0  01 00 0f 4c 69 6e 65 4e  75 6d 62 65 72 54 61 62  |...LineNumberTab|
000000e0  6c 65 01 00 04 6d 61 69  6e 01 00 16 28 5b 4c 6a  |le...main...([Lj|
000000f0  61 76 61 2f 6c 61 6e 67  2f 53 74 72 69 6e 67 3b  |ava/lang/String;|
00000100  29 56 01 00 0a 53 6f 75  72 63 65 46 69 6c 65 01  |)V...SourceFile.|
00000110  00 06 41 2e 6a 61 76 61  00 21 00 0e 00 02 00 00  |..A.java.!......|
00000120  00 00 00 03 00 01 00 05  00 06 00 01 00 19 00 00  |................|
00000130  00 1d 00 01 00 01 00 00  00 05 2a b7 00 01 b1 00  |..........*.....|
00000140  00 00 01 00 1a 00 00 00  06 00 01 00 00 00 01 00  |................|
00000150  09 00 11 00 12 00 01 00  19 00 00 00 1d 00 02 00  |................|
00000160  01 00 00 00 05 1a 10 0a  60 ac 00 00 00 01 00 1a  |........`.......|
00000170  00 00 00 06 00 01 00 00  00 03 00 09 00 1b 00 1c  |................|
00000180  00 01 00 19 00 00 00 28  00 02 00 01 00 00 00 0c  |.......(........|
00000190  b2 00 07 10 08 b8 00 0d  b6 00 13 b1 00 00 00 01  |................|
000001a0  00 1a 00 00 00 0a 00 02  00 00 00 07 00 0b 00 08  |................|
000001b0  00 01 00 1d 00 00 00 02  00 1e                    |..........|
000001ba
```

foo函数体内的指令

从 hexdump 显示的信息中，你能看到一些可以阅读的字符，比如“java/lang/Object”、"java/lang/System"等等，这些是常量表中的内容。还有一些内容显然不是字符，没法在屏幕上显示，所以 hexdump 就用一个. 号来表示。其中某些字节，代表的是指令，我在图中把代表 foo 函数、main 函数和构造函数的指令标注了出来，这些都是。用于运行的字节码指令，其他都是一些符号表等描述性的信息。

通过上图，你还能得到一个直观的印象：**字节码文件并不都是由指令构成的。**

没错，指令只是一个程序文件的一部分。除了指令以外，在字节码文件中还要存储不少其他内容，才能保证程序的正常运行，比如类和方法的符号信息、字符串和数字常量，等等。至于字节码文件的格式，是由字节码的规范来规定的，你有兴趣的话，可以按照规范生成这样的字节码文件。这样的话，我们的程序就可以在 JVM 上运行了。

不过，我现在不想陷入字节码文件格式的细节里，而是想用自己的方式生成字节码文件，够支持现在的语言特性，能够在我们自己的虚拟机上运行就行了。

上面这张图显示的字节码文件不是很容易阅读和理解。所以，我们用 javap 命令把它转化成文本格式来看看。

```
1 javap -v A.class > A.bc
```

复制代码

在这个新生成的文件里，我们可以清晰地看到每个函数的定义以及指令，我也在图里标注了主要的指令的含义。

文本格式的字节码文件

看到这个字节码文件的内容，你可能会直观地觉得：这看上去跟我们的高级语言也没有那么大的区别嘛。程序照样划分成几个函数，只不过每个函数里的语句变成了栈机的指令而已，函数之间照样需要互相调用。

实际上也确实没错。字节码文件里本来就存储了各个类和方法的符号信息，相当于保存了高级语言里的主体框架。当然，每个方法体里的代码就看不出 if 语句、循环语句这样的结构了，而是变成了字节码的指令。

通过研究这些指令，加上查阅JVM 规则中对于字节码的规定，你会发现为了实现上面示例代码中的功能，我们目前只需要这几个指令就够了：

助记符	操作码（十六进制）	操作码（二进制）	其他字节	描述
iload	15	0001 0101	1个字节：变量的序号	把本地变量入栈，操作数是变量的下标，参数也算作本地变量
iload_0	1a	0001 1010		把下标为0的本地变量入栈
...				...
iload_3	1d	0001 1101		把下标为3的本地变量入栈
istore	36	0011 0110	1个字节：变量的序号	把栈顶的值保存进指定下标的变量
istore_0	3b	0011 1011		把栈顶的值保存进下标为0的变量
...				...
istore_3	3e	0011 1110		把栈顶的值保存进下标为3的变量

极客时间

助记符	操作码（十六进制）	操作码（二进制）	其他字节	描述
iconst_0	3	0000 0011		把常数0入栈
...				...
iconst_5	8	0000 1000		把常数5入栈
bipush	10	0001 0000	1个字节：8位整数	把一个8位整数入栈
sipush	11	0001 0001	2个字节：16位整数	把1个16位整数入栈
ldc	12	0001 0010	1个字节：常量的序号	把一个常量入栈，该常量可能是String、int、float、Class等。
iadd	60	0110 0000		整数加法运算（从栈里弹出两个值，运算完毕后入栈，下面相同）

极客时间

常数入栈指令

助记符	操作码（十六进制）	操作码（二进制）	其他字节	描述
isub	64	0110 0100		整数减法运算
imul	68	0110 1000		整数乘法运算
idev	6c	0110 1100		整数除法运算
invokestatic	b8	1011 1000	两个字节：方法索引值	调用一个静态方法，并把结果放回栈。该方法用常量池中的索引值来查找。索引值是由两个字节拼成的：(indexbyte1 << 8 | indexbyte2)
ireturn	ac	1010 1100		弹出栈顶的一个整数并作为返回值返回
return	b1	1011 0001		返回void

极客时间

二元运算指令，以及函数调用和返回指令

你先花一两分钟看一下这些指令，看上去挺多，其实可以分为几组。

首先是 iload 系列，这是把指定下标的本地变量入栈。注意，变量的下标是由声明的顺序决定的，参数也算本地变量，并且排在最前面。所以，iload 0 的意思，就是把第一个参数入栈。如果没有参数，就是把第一个本地变量入栈。

iload 后面的那几个指令，是压缩格式的指令，也就是利用指令末尾富余的位，把操作数和指令压缩在了一起，这样可以少一个字节码，能够缩小最后生成的字节码文件的大小。从这里面，你能借鉴到字节码设计的一些好的实践。所以你看，学习成熟的设计是有好处的吧？

第二组是 istore 系列，它做的工作刚好跟 iload 相反，是把栈顶的值存到指定下标的变量里去。

第三组，是对常数做入栈的操作。对于 0~5 这几个数字，Java 字节码也是提供了压缩格式的指令。对于 8 位整数（-128~127），使用 bipush 指令。对于 16 位整数（-32768~32767），使用 sipush 指令。而对于更大的常数，则要使用 ldc 指令，从常量池里去取。

第四组，是几个二元运算的指令。它们都是从栈里取两个操作数，计算完毕之后，再压回栈里。

最后一组指令，是函数调用和返回的指令。函数调用的时候，也是从栈里取参数。返回值呢，则压回栈里。

通过这些指令，我们就完全能够实现一些基本的功能了，之后我们再根据需要添加更多的指令就好。

现在，我们把这些指令做成枚举值，方便程序使用：

复制代码

```
/**
 * 指令的编码
 */
enum OpCode{
    iconst_0= 0x03,
    iconst_1= 0x04,
    iconst_2= 0x05,
    iconst_3= 0x06,
```

```
    iconst_4= 0x07,
    iconst_5= 0x08,
    bipush  = 0x10,  //8位整数入栈
    sipush  = 0x11,  //16位整数入栈
    iload   = 0x15,  //本地变量入栈
    iload_0 = 0x1a,
    iload_1 = 0x1b,
    iload_2 = 0x1c,
    iload_3 = 0x1d,
    istore  = 0x36,
    istore_0= 0x3b,
    istore_1= 0x3c,
    istore_2= 0x3d,
    istore_3= 0x3e,
    iadd    = 0x60,
    isub    = 0x64,
    imul    = 0x68,
    idiv    = 0x6c,
    ireturn = 0xac,
    return  = 0xb1,
    invokestatic= 0xb8, //调用函数
}
```

好了，我们通过全盘照抄的方式，“设计”出了自己所需要的字节码。不过这些都只是指令的部分。还要有常量池的部分，我们在下面使用到的时候再去设计。

接下来，我们来生成自己的字节码。

生成字节码

生成栈机的字节码是比较简单的。为什么呢？因为栈机的字节码基本上跟 AST 是同构的，通过深度优先的顺序遍历 AST 就可以实现。相对来说，生成寄存器机的指令的算法就要稍微绕一点，我们也会在后面的课程中体会到。

怎么来深度优先地遍历 AST、生成栈机的字节码呢？以“3+a”这样一个简单的表达式为例，我们处理的顺序依次是字面量 3、变量 a 和 + 号。

首先来处理字面量。处理整型字面量的时候，我们需要根据整数的不同长度，分别使用不同的常量指令。其中的一个细节是，当整数是 32 位时，操作数要拆成两个字节。而当大于 32 位时，我们干脆就把字面量放在常量池里，操作数只是该常数在常量池的索引值。

这里的处理你可以参照下面这个代码：

复制代码

```
visitIntegerLiteral(integerLiteral: IntegerLiteral):any{
    let ret:number[] = [];
    let value = integerLiteral.value;
    //0-5之间的数字，直接用快捷指令
    if (value >= 0 && value <= 5) {
        switch (value) {
            case 0:
                ret.push(OpCode.iconst_0);
                break;
            ...省略1、2、3、4的情况
            case 5:
                ret.push(OpCode.iconst_5);
                break;
        }
    }

    //如果是8位整数，用bipush指令，直接放在后面的一个字节的操作数里就行了
    else if (value >= -128 && value <128){
        ret.push(OpCode.bipush);
        ret.push(value);
    }

    //如果是16位整数，用sipush指令
    else if (value >= -32768 && value <32768){
        ret.push(OpCode.sipush);
        //要拆成两个字节
        ret.push(value >> 8);
        ret.push(value & 0x00ff);
    }

    //大于16位的，采用ldc指令，从常量池中去取
    else{
        //把value值放入常量池。
        this.module.consts.push(value);
        ret.push(this.module.consts.length -1);
    }
    return ret;
}
```

接着来处理变量 a。在处理变量的时候，我们要区分左值和右值的情况。在“3+a”这个表达式中，a 是个右值，我们需要取出 a 的值，也就是要生成 iload 指令。但对于“a=3”这样的表达式，a 是个左值，这个时候返回 a 的符号即可，在处理赋值运算的时候再生成 istore 指令。

复制代码

```
/**
 * 左值的情况，返回符号。否则，生成iload指令。
 * @param v
 */
visitVariable(v:Variable):any{
    if (v.isLeftValue){
        return v.sym;
    }
    else{
        return this.getVariableValue(v.sym);
    }
}

/**
 * 生成获取本地变量值的指令
 * @param varName
 */
private getVariableValue(sym:VarSymbol|null):any{
    if (sym != null){
        let code:number[] = [];  //生成的字节码
        //本地变量的下标
        let index = this.functionSym?.vars.indexOf(sym);
        assert(index != -1, "生成字节码时（获取变量的值），在函数符号中获取本地变量下标
        //根据不同的下标生成指令，尽量生成压缩指令
        switch (index){
            case 0:
                code.push(OpCode.iload_0);
                break;
            case 1:
                code.push(OpCode.iload_1);
                break;
            case 2:
                code.push(OpCode.iload_2);
                break;
            case 3:
                code.push(OpCode.iload_3);
                break;
            default:
                code.push(OpCode.iload);
                code.push(index as number);
        }
        return code;
    }
}
```

然后，我们要对加减乘除这些二元运算来生成代码，我们可以先为左右子树分别生成代码，再把加减乘除的运算指令放在最后。注意，赋值运算的处理逻辑是不同的，它要生成 istore 指令。

复制代码

```
visitBinary(bi:Binary):any{
    let code:number[];
    let code1 = this.visit(bi.exp1);
    let code2 = this.visit(bi.exp2);

    ////1.处理赋值
    if (bi.op == Op.Assign){
        let varSymbol = code1 as VarSymbol;
        //加入右子树的代码
        code = code2;
        //加入istore代码
        code = code.concat(this.setVariableValue(varSymbol));
    }
    ////2.处理其他二元运算
    else{
        //加入左子树的代码
        code = code1;
        //加入右子树的代码
        code = code.concat(code2);
        //加入运算符的代码
        switch(bi.op){
            case Op.Plus: //'+'
                code.push(OpCode.iadd);
                break;
            case Op.Minus: //'-'
                code.push(OpCode.isub);
                break;
            case Op.Multiply: //'*'
                code.push(OpCode.imul);
                break;
            case Op.Divide: //'/'
                code.push(OpCode.idiv);
                break;
            default:
                console.log("Unsupported binary operation: " + bi.op);
                return [];
        }
    }
    return code;
}
```

好了，到此这里，我们对处理基本的表达式就没有问题了。接下来，我们再增加与函数调用和返回有关的指令。

在进行函数调用的时候，我们要依次生成与参数计算有关的指令，最后生成 invokestatic 指令。

复制代码

```
let code:number[] = [];
//1.依次生成与参数计算有关的指令
for(let param of functionCall.paramValues){
    let code = this.visit(param);
    if (typeof code == 'object'){
        code = code.concat(code as number[]);
    }
}
//2.生成invoke指令
let index = this.module.consts.indexOf(functionCall.sym);
code.push(OpCode.invokestatic);
code.push(index>>8);
code.push(index);
return code;
```

然后，在处理 return 语句的时候，也要注意，我们要根据是否有返回值，分别生成 ireturn 和 return 指令。

复制代码

```
let code:number[] = [];
//1.为return后面的表达式生成代码
if(returnStatement.exp != null){
    let code1 = this.visit(returnStatement.exp);
    if (typeof code1 == 'object'){
        code = code.concat(code1 as number[]);
        //生成ireturn代码
        code.push(OpCode.ireturn);
        return code;
    }
}
else{
    //2.生成return代码，返回值是void
    code.push(OpCode.return);
    return code;
}
```

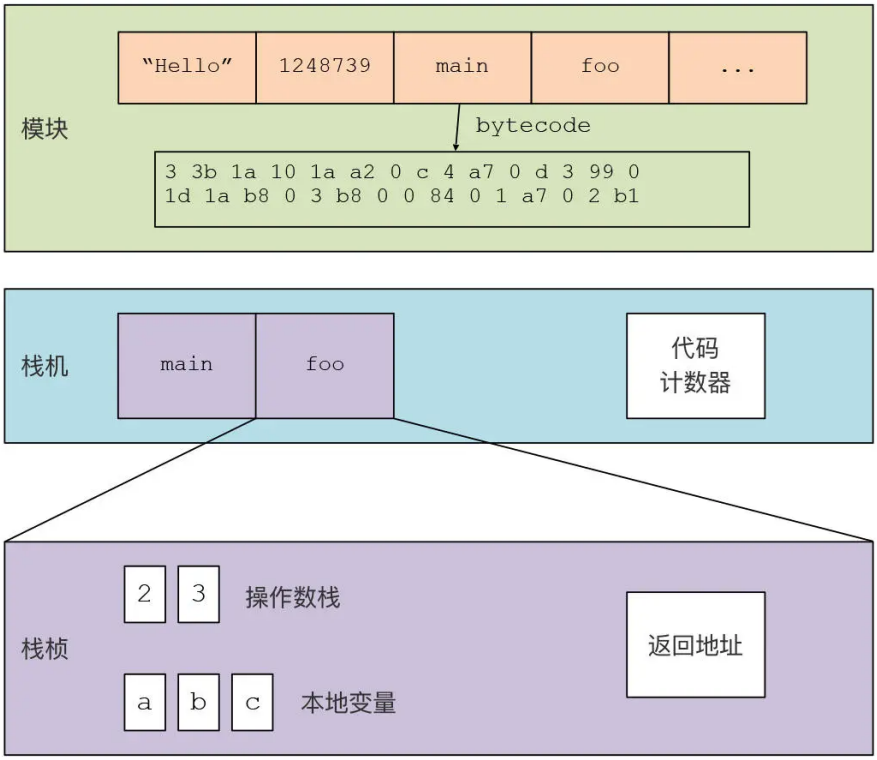
到这里，我们已经能够顺利的生成字节码了。生成字节码以后，接下来就只剩最后一步了：实现一个虚拟机，让这些字节码真正运行起来！

实现一个 TypeScript 版本的虚拟机

如果你是学习 Java 的同学，那你在面试的时候肯定经常会被问到与虚拟机有关的知识点。你也会根据自己了解的知识，对操作数栈、常量池等发表一通见解。但是，你心里多读少少对这些概念还会隔着一层面纱。

直到你自己亲自动手实现一遍，哪怕只是实现一个原型系统，你对这些概念，以及对虚拟机到底是如何运行的，才会有真真切切的理解。所以，我鼓励你动手实现一遍。而且，我可以告诉你，真的花不了多少时间。如果你看看我给出的示例代码，其实就是一个大函数，针对不同的字节码指令做处理而已。

不过，为了便于你理解代码，我还是先画一个虚拟机的示意图。我们这个简单的虚拟机主要涉及几个对象：



极客时间

首先是模块。模块代表了我们的程序，模块里最重要的就是一个常量表。常量表中的常量包含字符串常量、数字常量和函数符号。而函数符号里有一个 bytecode 属性，保存了这个函数的字节码。这样，我们就可以找到函数的代码并运行它们了。

复制代码

```
export class BCModule{
  //常量
  consts:any[] = [];

  //入口函数
  _main:FunctionSymbol|null = null;
}
```

然后是运行程序的栈机。栈机里最重要的数据结构是一个调用栈，在 AST 解释器里也有类似的数据结构。调用栈是由栈帧构成的，在执行每个函数的时候，都需要在栈顶新加一个栈帧，而在退出函数的时候，就会弹出这个函数的栈帧。

复制代码

```
//栈机
export class VM{
    //调用栈
    callStack:StackFrame[]=[];
    //执行一个模块
    execute(bcModule:BCModule):number{
    ...
    }
}
```

你要注意，栈帧是一个关键的数据结构，它用来保存每个函数运行时所需要保存的状态信息。每个栈帧里又有几个关键的组成部分。

操作数栈：用来保存函数执行过程中各个指令所需要用到的操作数；

存放本地变量的数组：可以在这里存取变量值；

返回地址：也是在调用子函数的时候，告诉子函数返回以后，从哪条代码接着运行。

栈机里的 execute 方法，就是虚拟机的核心执行引擎。下面我摘一些有代表性的代码给你讲一下，你先看一下示例代码的结构。

复制代码

```
//一直执行代码，直到遇到return语句
let opCode = code[codeIndex];
while(true){
    switch (opCode){
```

```
        case OpCode.iconst_0: //加载常量
            frame.oprandStack.push(0);
            opCode = code[++codeIndex];
            continue;
        ...
        case OpCode.sipush:  //加载32位常量，需要取出2个字节
            let byte1 = code[++codeIndex];
            let byte2 = code[++codeIndex];
            frame.oprandStack.push(byte1<<8|byte2);
            opCode = code[++codeIndex];
            continue;
        ...
        case OpCode.iload_0: //从变量里取值
            frame.oprandStack.push(frame.localVars[0]);
            opCode = code[++codeIndex];
            continue;
        ...
        case OpCode.istore_0: //给变量赋值
            frame.localVars[0] = frame.oprandStack.pop();
            opCode = code[++codeIndex];
            continue;
        ...
        case OpCode.iadd:     //加减乘除
            frame.oprandStack.push(frame.oprandStack.pop() + frame.oprandStack
            opCode = code[++codeIndex];
            continue;
        ...
    }
}
```

你可能马上会注意到，**整个程序执行的过程就是一个大的 while 循环。**

几乎所有的虚拟机的核心执行引擎都是这么写的，因为程序的执行过程就是不停地读取指令，然后根据指令做相应的动作。其中的代码计数器，一直指向下一个要执行的指令的位置。由于指令的种类比较多，所以 switch 后面会罗列很多个 case，代码也会很长。你用电脑看这些代码的话，可能要翻很多屏。

虽然这个函数的代码很长，我们一般也不会把它拆成多个函数。为什么呢？这似乎违背了通常的编程理念呀。通常我们都不会编写太长的函数，这不容易阅读，也不容易维护。

但事情总有例外。对于虚拟机的执行引擎来说，性能上的考虑是第一位的。执行一个指令，可能开销并不大。但如果执行这条指令要调用一个专门的函数，那函数调用的额外开

销会比执行指令本身的开销都大，那显然就不合理了，这也是我们为什么弃用 AST 解释器的原因。

我们这个示例代码中体现了加载常量、加载变量、保存变量和执行加减乘除的运算的实现方式。你能通过这些代码，再一次验证栈机的工作原理，这里所有操作都是围绕着操作数栈来进行的。

而函数的调用和返回，则显得复杂一点。这两部分代码，也有助于你更加细致的了解栈桢的使用方式。

首先我们来看看函数调用：

复制代码

```
case OpCode.invokestatic:
    //从常量池找到被调用的函数
    byte1 = code[++codeIndex];
    byte2 = code[++codeIndex];
    let functionSym = bcModule.consts[byte1<<8|byte2] as FunctionSymbol;

    //设置返回值地址，为函数调用的下一条指令
    frame.returnIndex = codeIndex;

    //创建新的栈桢
    let lastFrame = frame;
    frame = new StackFrame(functionSym);
    this.callStack.push(frame);

    //传递参数
    let paramCount = (functionSym.decl as FunctionDecl).callSignature.params.l
    for(let i = paramCount -1; i>= 0; i--){
        frame.localVars[i] = lastFrame.oprandStack.pop();
    }

    //设置新的code、codeIndex和oPCode
    if (frame.funtionSym.bytecode !=null){
        //切换到被调用函数的代码
        code = frame.funtionSym.bytecode;
        //代码指针归零
        codeIndex = 0;
        opCode = code[codeIndex];
        continue;
    }
```

在函数调用的时候，我们要完成这几项工作：

设置返回地址，被调用的函数在执行 return 指令的时候，会回到这个位置；

为被调用的函数生成新的栈桢，并加到调用栈中；

从操作数栈中取出参数的值，并赋给新的栈桢，因为参数也算作本地变量，因此只需要对新栈桢的前几个本地变量赋值就可以了；

把代码切换到被调用函数的代码，并且把代码计数器设置为 0，也就是从第一个字节码指令开始执行。

而函数返回是函数调用的逆向操作：

复制代码

```
case OpCode.ireturn:
case OpCode.return:
    //确定返回值
    let retValue = undefined;
    if(opCode == OpCode.ireturn){
        retValue = frame.oprandStack.pop();
    }

    //弹出栈桢，返回到上一级函数，继续执行
    this.callStack.pop();
    if (this.callStack.length == 0){ //主程序返回，结束运行
        return 0;
    }
    else { //返回到上一级调用者
        frame = this.callStack[this.callStack.length-1];
        //设置返回值到上一级栈桢
        // frame.retValue = retValue;
        if(opCode == OpCode.ireturn){
            frame.oprandStack.push(retValue);
        }
        //设置新的code、codeIndex和oPCode
        if (frame.funtionSym.byteCode !=null){
            //切换到调用者的代码
            code = frame.funtionSym.byteCode;
            //设置指令指针为返回地址，也就是调用该函数的下一条指令
            codeIndex = frame.returnIndex;
            opCode = code[codeIndex];
            continue;
        }
        else{
            console.log("Can not find code for "+ frame.funtionSym.name);
            return -1;
```

```
        }
    }
    continue;
```

函数返回时，我们需要完成下面这几项工作：

设置返回值，如果确实有一个返回值，那么就从当前操作数栈中取出来，返回值加载到上一级栈桢里的操作数栈；

从调用栈中弹出当前栈桢；

把代码切换成调用者的代码，并且把代码指针设置成函数调用指令的下一条指令，继续函数调用之后的工作；

这里还有一个特殊情况，执行到 return 指令的时候，你会发现当前栈桢之上，再也没有上一级栈桢了。这说明当前所在的函数已经是最顶层了，也就是我们的编译器内部的 _main() 函数。这个时候，return 指令就会结束整个 while 循环，也意味着整个程序运行结束。

好了，这就是函数调用和返回的过程。通过这个过程，你会发现这两类指令其实是遵循了一些共同的约定。

这些约定包括：我们传参数的时候，把参数放在哪里？函数返回的时候，又把返回值放在哪里？还有，调用函数的时候，我们应该在一个双方都知道的位置设置返回地址，以便函数返回后调到这个地址继续执行。

这里，你又学到一个概念，叫做**调用约定（Calling Convention）**。计算机语言的设计者，可以设计自己的调用约定，这样用自己的编译器编译出来的模块，都能互相调用。但如果一种语言想调用另一种语言编写的模块，那么它们必须遵循相同的调用约定，或者要在不同的调用约定之间做转换。比如，Java 调用 C 语言写的模块，要使用 JNI 接口，就是要完成这种调用约定的转换。

跟调用约定差不多的一个概念叫做 **ABI（Application Binary Interface）**。ABI 这个名字更强调如何在二进制的层面上实现互相调用，以及二进制程序文件的格式，等等，所以更适合描述像 C、C++ 这些编译成二进制目标代码的情形。

好了，现在我们已经拥有了一个简化版的虚拟机了，你可以用它跑几个程序试一下。因为现在我们这个虚拟机还不支持 if 语句和循环语句，所以还不方便做性能测试。这个工作，我们放在下一节课再去做。

课程小结

好了，今天这节课到这里就结束了，让我们来简单回顾一下。

今天，我们设计并实现了一个简单的基于栈机的虚拟机，实现了我们在虚拟机领域 0 的突破。如果你认真跟着学完了这一节，我相信你一定会有很多收获。

首先，我们学了栈机的指令，通常包括加载常数、加载变量、保存变量、加减乘除、函数调用和返回这些，用这些指令就能让程序运行起来了。下一节课我们还会多学习一些指令，特别是分支指令。

第二，我们已经可以通过遍历 AST 的方式生成字节码。其中稍微有点难度的地方，是在访问变量节点的时候，要区分左值和右值，分别生成给变量赋值的代码和读取变量值的代码。

第三，在虚拟机里，我们用了 BCModule 这个数据结构来表示可被执行的程序。其实你把这个数据结构保存到文件里就是字节码文件了，它里面主要的内容就是一个常量表。常量表里除了字符串、数字这样的常规意义上的常数以外，最重要的还有函数符号，函数符号中包含了要运行的字节码。

第四，栈机的执行引擎是一个很大的循环，要依次执行每条指令，在这个过程中操作数栈会不停的压入数据、弹出数据。在调用函数和返回函数的时候，要建立栈桢和弹出栈桢。函数参数被设置到新栈桢的本地变量里，而返回值则被设置到上一级栈桢的操作数栈里，这就构成了函数之间的调用约定。

虽然我们目前是在虚拟机层面上了解这些概念，但是，即使是到了我们把程序编译成机器码的时候，这些基本概念仍然是差不多的。你现在学习的成果，会为后面的学习内容打下很好的基础。

在下一节课，我们将继续深化我们虚拟机。到时候它会在哪些方面取得突破呢？敬请期待吧！

思考题

在这节课的 BCModule 中有一个常量表，为什么把函数符号也看做是常量呢？刨去其中的字节码，这些符号信息可能有什么潜在的用途呢？欢迎在留言区发表你的看法！

感谢你和我一起学习，欢迎你把我这节课分享给更多对字节码虚拟机感兴趣的朋友。我是宫文学，我们下节课见！

课程资源

这节课的示例代码在这里！

分享给需要的人，Ta订阅后你可得 20 元现金奖励

赞 0 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 07 | 怎么设计属于我们自己的虚拟机和字节码？

下一篇 09 | 基于TypeScript的虚拟机（二）：丰富特性，支持跳转语句

精选留言 (2)

写留言

奋斗的蜗牛
2021-08-26

回答一下思考题，符号表信息可以实现函数式编程的支持

bjuthang
2021-08-25

赞

展开